# MyBatis概述

### 概述

MyBatis是最近几年非常流行的数据访问层(DAO)框架，能够简单高效的实现对数据层访问。

### 常见数据访问层方式比较

#### JDBC

java原生的关系型数据库访问方式

* 每次操作数据库都需要获取连接关闭连接，在大量访问数据库时，频繁的开关连接消耗性能。
* 需要手动编写sql，有学习成本
* 查询出的结果需要手动进行封装到bean
* 没有缓存处理机制
* sql语句写死在程序中，需要修改sql必须修改源文件

#### Hibernate

基于面向对象理念设计的DAO层框架，基本理念就是维护对象到表的映射关系，通过操作对象操作表中的数据，从而可以减少甚至杜绝sql的使用

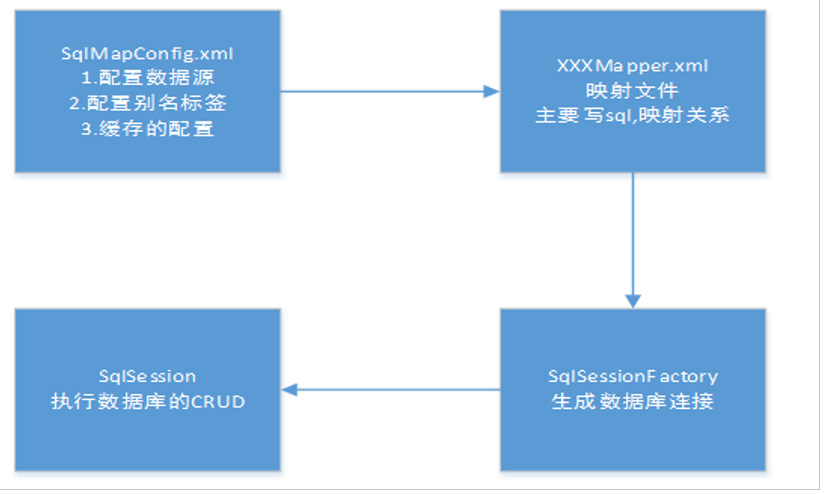
* 相对比较沉重，效率不好
* 当涉及到比较复杂的查询时Hibernate的操作对象的方式用起来非常麻烦，甚至无法实现，只能用sql操作
* 底层需要频繁的拼接sql，产生大量冗余的sql

#### MyBatis

是一种半自动对象-表映射关系的DAO层框架，可以自动的进行对象的封装，但是sql仍然需要自己来写。

* 结合了JDBC和Hibernate的优点
* 可以手写sql灵活实现数据访问
* 自动封装数据
* 减少冗余代码
* 执行效率比JDBC低

# MyBatis的结构



# MyBatis结构的创建

### 创建XXXMapper.xml文件

在该文件下配置sql语句，供程序调用

例如：创建一个user相关的UserMapper.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  //namespace为命名空间 <mapper namespace="cn.shu.blog.dao.UserDaoInter">  <!--用户登录-->  <!—这里的#{}代表可以接收参数 -->  <select id="userLogin" resultType="cn.shu.blog.beans.User">  SELECT \* FROM users WHERE account=#{userName} and password=#{password}  </select> </mapper> |

### 创建SqlMapConfig.xml文件

Mybatis的核心配置文件,在Src目录下创建该文件

注意：根标签<configuration>

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <!--可以配置多个数据源，这里指定默认值-->  <environments default="MySQL">  <environment id="MySQL">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql:///myblog"/>  <property name="username" value="admin"/>  <property name="password" value="pws"/>  </dataSource>  </environment>  </environments>  <mappers>  <!--指定SQL映射文件 可以写多个-->  <mapper resource="mappers/userMapper.xml"/>  </mappers> </configuration> |

### 创建连接工厂

### 获取连接

|  |
| --- |
| public static void main(String[] args) throws IOException {  //1、加载配置文件  InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");  //2、创建连接工厂  *SqlSessionFactory* build = new SqlSessionFactoryBuilder().build(inputStream);  //3、获取连接  *SqlSession* sqlSession = build.openSession();  //4、操作数据库  *Map*<String,String> map=new HashMap<>();  map.put("userName","shuxinsheng");//多个参数用map传递  map.put("password","123");  sqlSession.selectOne("cn.shu.blog.dao.UserDaoInter.userLogin",map);  //5、关闭连接  sqlSession.close(); } |

SqlSession有很多方法，第一个参数为SQL的命名空间 . SQL ID

第二个参数为传递给SQL的参数

# MyBatis接口方式使用

在上面的例子中，调用查询方法，至少需要传递一个String参数(SQL语句的命名空间 . SQL ID),另一个可选参数即为传递给SQL的参数:

|  |
| --- |
| sqlSession.selectOne("cn.shu.blog.dao.UserDaoInter.userLogin",map); |

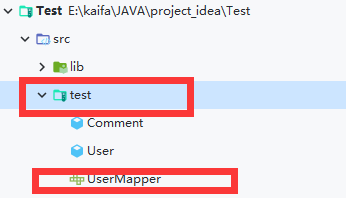
他们的缺点还是挺明显的，命名空间.SQLID名字太长，且容易写错，不易维护，可读性差。其次是传递的参数类型有限制。

为了简化MyBatis的使用，MyBatis提供了接口方式自动生成调用过程的机制，可以大大简化MyBatis的开发。

### 创建SQL映射文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <mapper namespace="test.UserMapper"> <select id="userLogin" resultType="test.User" >  SELECT \* FROM users where account =#{account} </select> </mapper> |

### 创建接口(接口的全路径名与命名空间一致)



### 创建方法(方法名与SQL的ID一致)

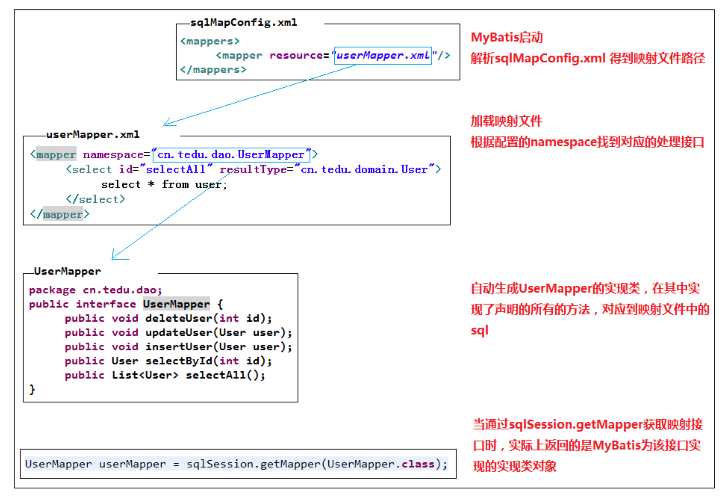
### 声明方法参数(方法参数与SQL中的参数一致)

### 声明方法返回值（方法返回值与SQL中声明的返回值一致）

|  |
| --- |
| public interface *UserMapper* {  //接口方法  public User userLogin( String account); } |

### 使用

|  |
| --- |
| //1、加载配置文件 InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml"); //2、创建连接工厂 *SqlSessionFactory* build = new SqlSessionFactoryBuilder().build(inputStream); //3、获取连接 *SqlSession* sqlSession = build.openSession(); //4、获取对象 MyBatis会自动创建该类的实现类并返回 *UserMapper* userMapper=sqlSession.getMapper(*UserMapper*.class); //5、执行SQL userMapper.userLogin("123"); //6、关闭连接 sqlSession.close(); |



# MyBatis的增删改查

### Insert

通过Trim拼接，prefix表示拼接完成后以什么开头，suffix以什么结尾, suffixOverrides表示什么分隔，他会去掉指定的最后一个参数的逗号

Test指定表达式，值为True则拼接

|  |
| --- |
| <insert id="myInsert">  insert into user  <trim prefix="(" suffix=")" suffixOverrides=",">  id,  <if test="name!=null">name,</if>  </trim>  values  <trim prefix="(" suffix=")" suffixOverrides=",">  id,  <if test="name!=null">#{name},</if>  </trim> </insert> |

### Delete

|  |
| --- |
| <delete id="myDelete">  delete from myblog.users  <where>  <if test="name!=null">name=#{name}</if>  </where> </delete> |

也可通过foreach遍历 然后拼接where

Collection 传递的集合名

Open 以什么开头

Close 以什么结尾

Separator 以什么分隔

最终效果 in ( “shu”,”xin” “sheng”)

|  |
| --- |
| <delete id="myDelete">  delete from myblog.users where name in  <foreach collection="names" open="(" close=")" separator="," item="name">  #{name}  </foreach> </delete> |

### Update

Set标签会自动去掉最后一个逗号

|  |
| --- |
| <update id="myUpdate">  update user  <set>  <if test="name!=null">name=#{name},</if>  </set>  <where>  <if test="name!=null">name=#{name}</if>  </where> </update> |

### Select

**查询结果可以封装到Bean中，需要通过resultType 指定类的全路径名**

|  |
| --- |
| <select id="userLogin" resultType="test.User" >  SELECT \* FROM users  <where>  <if test="account!=null">account =#{account}</if>  </where> </select> |

**如果查询结构的列名和Bean属性不一致，可以手动指定映射关系**

|  |
| --- |
| <resultMap id="re" type="test.User">  <!--主键列 必须配置-->  <id property="id" column="userId"/>  <!--其它列-->  <result property="name" column="userName"/> </resultMap> <select id="userLogin" resultMap="re">  SELECT \* FROM users  <where>  <if test="account!=null">account =#{account}</if>  </where> </select> |

### 注意事项

#### 上面的foreach set where等标签可以根据情况在不同类型的SQL语句中使用

#### 普通方式的Insert、Delete、Update，操作完成记得提交事务

# 参数传递方式

上面的例子中传递了Map参数，参数传递的方式有多种。

首先是在sql语句中用#{}或${}指定接收的参数名，然后在调用方法时传递参数即可

### #{}与${}传值的区别

#### #{}

* 相当于 PreParedStatement
* #{}作为方法参数，替代SQL中的?
* #{}预编译，效率高、防止sql注入
* #{}会在字符串类型二边添加单引号

#### ${}

* 会原样拼接到SQL上
* 没有预编译、效率低、不能防止SQL注入
* 字符串类型二边不会拼接单引号
* 在order by等其它语句中指定字段名，不需要单引号，所以就必须用这个

### 单值传递

#### XML:resultType中指定返回的数据类型为User对象

|  |
| --- |
| <!--用户登录--> <select id="userLogin" resultType="User">  SELECT \* FROM users where account =#{account} </select> |

#### JAVA：传递参数 1 赋值给sql语句的#{}位置

|  |
| --- |
| User user=sqlSession.selectOne("cn.shu.blog.dao.test.userLogin","1"); System.*out*.println(user); |

#### 注意，单值传递如果需要if判断，不能用参数名

|  |
| --- |
| <!--用户登录--> <select id="userLogin" resultType="User">  SELECT \* FROM users  <if test="account!=null">//报错  where account =#{account}  </if> </select> |

这里会报错，因为单值传递没有指定参数名

|  |
| --- |
| There is no getter for property named 'account' in 'class java.lang.String' |

！！！ 需要用 \_parameter

|  |
| --- |
| <!--用户登录--> <select id="userLogin" resultType="User">  SELECT \* FROM users  <if test="\_parameter!=null">  where account =#{account}  </if> </select> |

### 顺序传参

这种方式只能通过MyBatis的接口方式使用，普通方式只能接收一个Object参数

即Mapper接口的参数有多个，无需@param指定，然后再xml文件中通过序号接收

#### 接口方法

|  |
| --- |
| public User userLogin(String account,String password); |

#### Xml文件

|  |
| --- |
| <select id="userLogin" resultType="test.User">  SELECT \* FROM users where account=#{account}  and password=#{password} </select> |

#### 使用问题

因为按顺序传递，xml文件的SQL语句只能通过序号接收，或者在接口方法用@Param (“name”)注解指定参数名,所以报错

|  |
| --- |
| Parameter 'account' not found. Available parameters |
| *UserMapper* mapper = sqlSession.getMapper(*UserMapper*.class); //报错Parameter 'account' not found. Available parameters User user = mapper.userLogin("123", "123"); |

#### 问题解决：

##### @Param修饰

|  |
| --- |
| public User userLogin(@Param("account") String account, @Param("password")String password); |

|  |
| --- |
| <select id="userLogin" resultType="test.User">  SELECT \* FROM users where account=#{account} and password=#{password} </select> |

##### 序号(0 1 …)

|  |
| --- |
| <select id="userLogin" resultType="test.User">  SELECT \* FROM users where account=#{0} and password=#{1} </select> |

##### 参数序号(param1…)

|  |
| --- |
| <select id="userLogin" resultType="test.User">  SELECT \* FROM users where account=#{param1} and password=#{param2} </select> |

### Map传参

在SQL语句中通过，Map写的的KEY来接收参数

创建Map

|  |
| --- |
| *Map*<String,String> map=new HashMap<>(); map.put("account","123"); map.put("password","123"); |

#### 普通方式

|  |
| --- |
| User user = sqlSession.selectOne("test.UserMapper.userLogin", map); |

#### 接口方式

接口方法：

|  |
| --- |
| //接口方法  public User userLogin(*Map* map); |

sql：

|  |
| --- |
| <select id="userLogin" resultType="test.User">  <!—account和password为map的key-->  SELECT \* FROM users where account=#{account} and password=#{password} </select> |

调用：

|  |
| --- |
| *UserMapper* mapper = sqlSession.getMapper(*UserMapper*.class); User user = mapper.userLogin(map); |

### bean传参

即传递一个Bean对象，SQL语句中通过指定每个属性名获取参数

经测试，无需get或set方法一样可以传递参数

#### 普通方式

|  |
| --- |
| User loginUser = new User("guest","guest"); User user = sqlSession.selectOne("test.UserMapper.userLogin", loginUser); |

#### 接口方式

接口方法：

|  |
| --- |
| //接口方法  public User userLogin(User user); |

使用：

|  |
| --- |
| //调用  User loginUser = new User("guest","guest"); User user=sqlSession.getMapper(*UserMapper*.class).userLogin(loginUser); |

### JSON传参

SQL语句通过JSON的key查找对应值，没找到为null，查询结果应该也为空

需要引入JSON相关JAR包

|  |
| --- |
| //生成JSON String json="{\"account\":123,\"password\":123}"; JSONObject jsonObject = JSONObject.*parseObject*(json); |

#### 普通方式传参

|  |
| --- |
| User user1 = sqlSession.selectOne("test.UserMapper.userLogin", jsonObject); |

#### 接口方式传参

|  |
| --- |
| //接口方法 public User userLogin(JSONObject jsonObject); |

使用：

|  |
| --- |
| User user2=sqlSession.getMapper(*UserMapper*.class).userLogin(jsonObject); |

### 集合类型传递参数(List、Set、Array)

在SQL中foreach遍历，读取

|  |
| --- |
| //创建集合 *List*<String> arrayList = new ArrayList<>(); arrayList.add("1"); arrayList.add("123"); arrayList.add("123456"); |

SQL:

|  |
| --- |
| <!—-SQL语句-->  <select id="userLogin" resultType="test.User" >  SELECT \* FROM users where account in  <foreach collection="list" item="ac" separator="," open="(" close=")">  #{ac}  </foreach> </select> |

#### 普通方式

|  |
| --- |
| *List*<User> user8 = sqlSession.selectList("test.UserMapper.userLogin",arrayList); |

#### 接口方式

|  |
| --- |
| //接口方法 public *List*<User> userLogin(*List*<String> list); |

|  |
| --- |
| *//使用*  *List*<User> user9=sqlSession.getMapper(*UserMapper*.class).userLogin(arrayList); |

#### Foreach

|  |
| --- |
| foreach元素的属性主要有 item，index，collection，open，separator，close。  item表示集合中每一个元素进行迭代时的别名，  index指定一个名字，用于表示在迭代过程中，每次迭代到的位置，  open表示该语句以什么开始，  separator表示在每次进行迭代之间以什么符号作为分隔符，  close表示以什么结束  在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有一下3种情况：  1.如果传入的是单参数且参数类型是一个List的时候，collection属性值为list  2.如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array  3.如果传入的参数是多个的时候，我们就需要把它们封装成一个Map或者Object |

### (对象+集合)传参或者(对象+对象)

Bean对象:

|  |
| --- |
| public class User {  private Comment comment; }  public class Comment {  private String comm="1"; } |

这种方式应该只能接口方式，因为需要@Param指定参数名

|  |
| --- |
| //接口方法 public User userLogin(@Param("user") User user8); |

同理如果user对象的comment对象是集合，同样可以foreach遍历

|  |
| --- |
| <!--  user.comment.comm  获取bean参数 user对象  再获取其属性comment对象  最后获取comment对象的comm属性  --> <select id="userLogin" resultType="test.User" >  SELECT \* FROM users where account =#{user.comment.comm} </select> |

# 返回值的类型

### 返回一般数据类型

|  |
| --- |
| <!--  指定 resultType 返回值类型时 String 类型的，  string 在这里是一个别名，代表的是 java.lang.String  对于引用数据类型，都是将大写字母转小写，比如 HashMap 对应的别名是 'hashmap'  基本数据类型考虑到重复的问题，会在其前面加上 '\_'，比如 byte 对应的别名是 '\_byte'  -->  <select id="getEmpNameById" resultType="string">  select username from t\_employee where id = #{id}  </select> |

# MyBatis多表查询

### 一对一

Comments：

|  |
| --- |
| public class Comments {  private long id;  private String comment;  private Users users; } |

Users：

|  |
| --- |
| public class Users {  private long id;  private String nickname; } |

在通过MyBatis实现一对一查询时，需要通过resultMaps指定如何将结果集中的列名对应到目标bean中，在一对一的bean中，如果包含了另一个表的对应对象，则可以在resultMap标签中通过association标签来声明映射方式.

|  |
| --- |
| <!--获取文章评论--> <resultMap id="comments" type="cn.shu.bean.Comments">  <!--把数据库返回字段的commentId赋值给Comments对象的id属性-->  <id property="id" column="commentId"/>  <!--把数据库返回字段的commentId赋值给Comments对象的id属性-->  <!--如果不写，数据库返回的字段不会赋值给对象的属性，会是null-->  <result column="comment" property="comment"/>  <!-- property="users" 即comments中的users属性-->  <association property="users" javaType="cn.shu.bean.Users">  <!--把数据库返回字段的userId赋值给Comments对象下Users对象的id属性-->  <id column="userId" property="id"/>  <!--把数据库返回字段的nickname赋值给Comments对象下Users对象的nickname属性-->  <result column="nickname" property="nickname"/>  </association> </resultMap> <select id="getArticleComments" resultMap="comments">  select comments.id as commentId,users.id as userId,comment,nickname from comments,users  where comments.userId=users.id </select> |

这里本质上不是数据库的一对一关系，但是一个评论只对应一个用户，可以用这种方式查询

### 一对多

文章的**分类**与**文章**是**一对多**关系，一个分类对应多篇文章，一篇文章只能一个分类

Category：

|  |
| --- |
| public class Category {  private long id;  private String categoryName;  //一个分类 对应多篇文章  private *List*<Articles> articlesList; } |

Articles：

|  |
| --- |
| public class Articles {  private long id;  private String title;  //一篇文章 只有一个分类  private Category category; } |

**数据库查询结果：**

|  |
| --- |
|  |

**这里以Category为视角查询，则一个Category下面包含多个Articles对象(一对多)**

**此时需要用 collection标签指定articlesList属性为Articles类型，并设置对应关系**

**指定类型时用ofType,注意: association用的javaType**

|  |
| --- |
| <resultMap id="resultMapId" type="cn.shu.bean.Category">  <id property="id" column="categoryId"/>  <result property="categoryName" column="categoryName" />  <!--这里用的ofType-->  <collection property="articlesList" ofType="cn.shu.bean.Articles">  <id column="articlesId" property="id"/>  <result column="title" property="title"/>  </collection>  </resultMap> <select id="getArticleComments" resultMap="resultMapId">  SELECT articles.id as articlesId,category.id as categoryId,title,categoryName  FROM articles,category  WHERE category.id=articles.categoryId </select> |

如果以**文章**视角查询，那么一个**文章**对应一个**分类**，那么就回到了**一对一的**查询方式

### 多对多

无论从哪一方的**视角**查询，分开来看，都可以按照**一对多**的方式查询

# MyBatis的其它标签

### 别名标签

映射文件中大量使用类名长的JAVA类型，可以建立别名，需要的地方引用即可

sqlMapConfig文件中：

注意:该标签要放到最前面，configuration内的前面,但不能在settings前面

|  |
| --- |
| <typeAliases>  <typeAlias type="cn.shu.blog.beans.Article" alias="article"/> </typeAliases> |

映射文件中：代替长串包名

|  |
| --- |
| <select id="asd" resultType="article">  select \* from user </select> |

### Sql的复用

某段SQL重复出现，可以提取出来，到处引用

|  |
| --- |
| <sql id="testSql">  select \* from user </sql> <select id="asd" resultType="article">  <include refid="testSql"/> where id=10 </select> |

# MyBatis的缓存

缓存机制可以减轻数据库的压力，原理是在第一次查询时，将查询结果保存起来，后续查询相同的SQL，不是真的去查数据库，而是直接返回的第一次查询后保存的值。

虽然降低了数据库的压力，但同时不能及时获取最新的数据。

第三方工具Redis也可实现缓存

### 一级缓存

只在一个事务中有效：即在一个事务中先后执行多次同一查询，只有第一次去查。

而如果是不同事务执行相同SQL并不会共用缓存。

MyBatis一级缓存默认开启，且无法手动关闭

### 二级缓存

缓存全局有效，一个事务查询一个SQL得到结果，其它事务查询相同SQL，得到之前缓存的结果

作用范围大，时间长，可能造成危害更大，开发中很少使用。

MyBatis二级缓存默认关闭

#### 开启二级缓存

**a可在SQL映射文件中开启**

|  |
| --- |
| <mapper>  <cache/> </mapper> |

**B也可在MyBatis配置文件sqlMapConfig.xml中开启**

**注意：该方式需放在configuration标签下，且必须位于第一行，不然报错**

|  |
| --- |
| <configuration>  <!—开启二级缓存 -->  <settings>  <setting name="cacheEnabled" value="true"/>  </settings> </configuration> |